Übungen zu Algorithmen und Datenstrukturen für Medizintechnik
Wintersemester 2022/2023
Jonas Auernheimer, Alina Schüller

Lehrstuhl für sensorische Neurotechnologie
(https://www.neurotech.tf.fau.eu)

Aufgabenblatt 5 vom 21.11.2022, Abgabe am 11.12.2022

Aufgabe 5.1: Theorie

Punkte siehe StudOn
OOP, UML

Aufgabenstellung und Abgabe (individuell, nicht als Gruppe!) im StudOn.

Aufgabe 5.2: Snake

60 Punkte
Objektorientierte Programmierung

In dieser Aufgabe werden Sie den Spieleklassiker *Snake*[1] programmieren. Der Spieler kontrolliert in diesem Spiel eine sich ohne Unterbrechung bewegende Schlange. Ihr Ziel ist es, möglichst viele auf dem Spielfeld verteilte Äpfel zu fressen, ohne dabei mit einer Wand oder mit sich selbst zusammenzustoßen.

Hinweis: Entnehmen Sie alle benötigten Informationen zu Attributsnamen, Sichtbarkeiten, Schnittstellen, Klassenbeziehungen, etc. dem UML-Klassendiagramm in Abbildung 1!

1. Legen Sie ein neues Projekt `Snake` an.
2. **SnakeGame**
   In der Klasse `SnakeGame` sollen die Hauptfunktionalitäten des Spiels implementiert werden.

   a) Wir stellen Ihnen für diese Aufgabe eine minimale Basisklasse `AudGameWindow` zur Verfügung, die einige grundlegende Funktionen wie das Anzeigen des Spielefensters zur Verfügung stellt. Sie ist in der Datei `05-material.zip` enthalten, die Sie auf unserer StudOn-Seite herunterladen, entpacken und die Java-Datei dann in Ihr Projekt einbinden können.
   Achtung: Diese Klasse darf nicht verändert werden!

   b) Leiten Sie von der Klasse `AudGameWindow` eine neue Klasse `SnakeGame` ab. Lassen Sie IntelliJ alle geerbten abstrakten Methoden überschreiben. Diese neuen Methoden können Sie für den Moment leer lassen.

   Graphics und Color
   Informieren Sie sich in den Tafelübungs-Folien sowie in der Java-Dokumentation im Internet über die Klassen `java.awt.Graphics` und `java.awt.Color`!
   Aufgrund der technischen Einschränkungen von StudOn dürfen Sie nicht direkt auf diese Klassen zugreifen sondern nur auf die bereitgestellten Klassen AudGraphics und AudColor, die alle wichtigen Methoden auch bereitstellen und genauso verwendet werden können.

   c) Legen Sie außerdem eine `main`-Methode an. Erstellen Sie darin eine neue Instanz von `SnakeGame` und rufen Sie die (vererbte) Methode `start()` dieses Objektes auf.

[1] https://de.wikipedia.org/wiki/Snake_(Computerspiel)

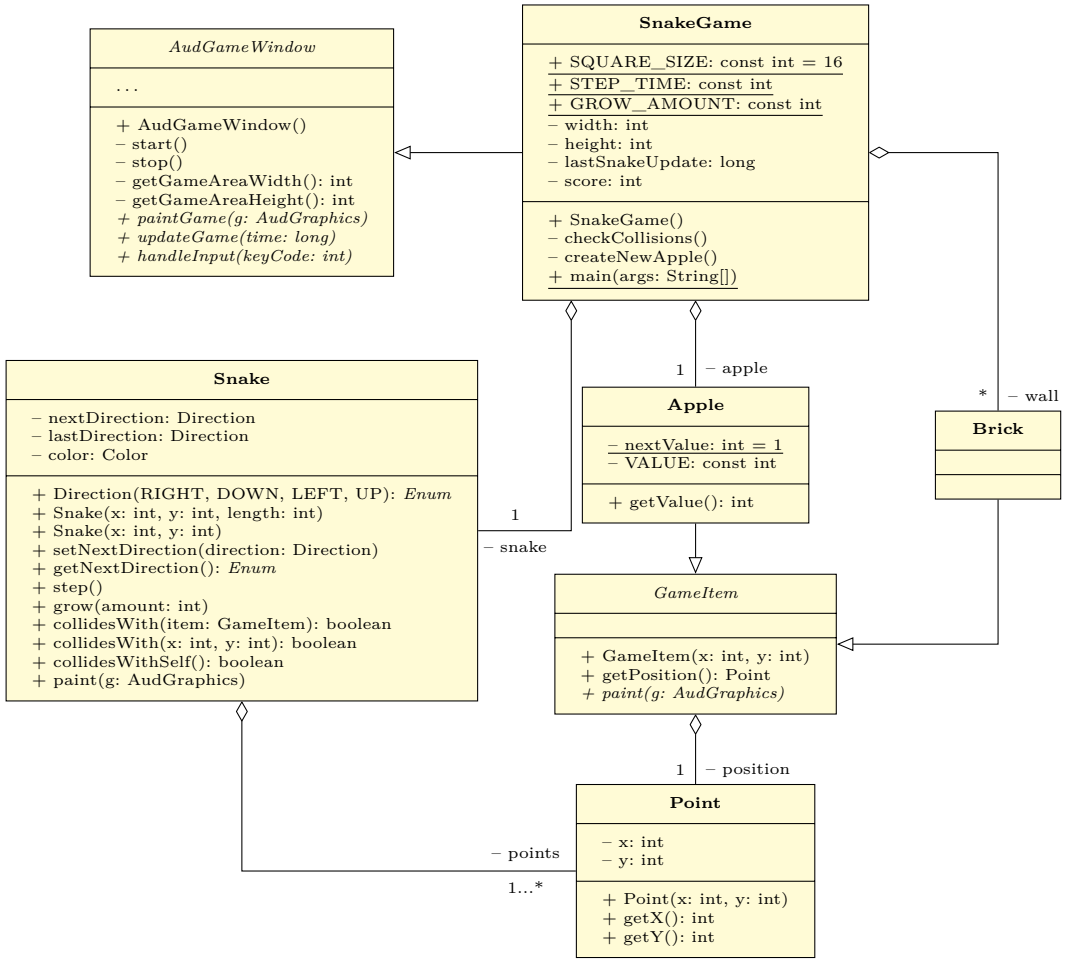


Abbildung 1: UML-Klassendiagramm des `Snake`-Projektes

Hinweis: Wenn Sie das Programm ausführen, sollten Sie jetzt schon einmal ein leeres Fenster wie in Abbildung 2 sehen.

d) In der Methode `void paintGame(AudGraphics g)` wird das Spiel gezeichnet. Zeichnen Sie dort zunächst ein ausgefülltes Rechteck als Hintergrund. Die dafür benötigte Größe des Spielbereiches können Sie mithilfe der Methoden `getGameAreaWidth()` und `getGameAreaHeight()` abfragen.

e) Legen Sie in `SnakeGame` einen Standard-Konstruktor an.
Mit der Methode `setTitle(String)` können Sie dort den Titel des Spielfensters ändern. Setzen Sie den Titel wie folgt:

```
AuD-Snake - Score: ⟨score⟩
```

⟨*score*⟩ (also der aktuelle Punktestand) soll durch den Wert des Attributes `score` ersetzt werden. Zu Spielbeginn ist der Punktestand (und damit der Wert von `score`) natürlich noch 0.

f) Der Einfachheit halber werden alle Spielobjekte in ein quadratisches Gitter eingepasst. Die Schlange bewegt sich also beispielsweise in einem »Schritt« um ein ganzes *Quadrat* weiter, nicht nur um einen *Pixel*. Im Folgenden werden wir dieses Gitter als »Gitter-Koordinatensystem« bezeichnen.

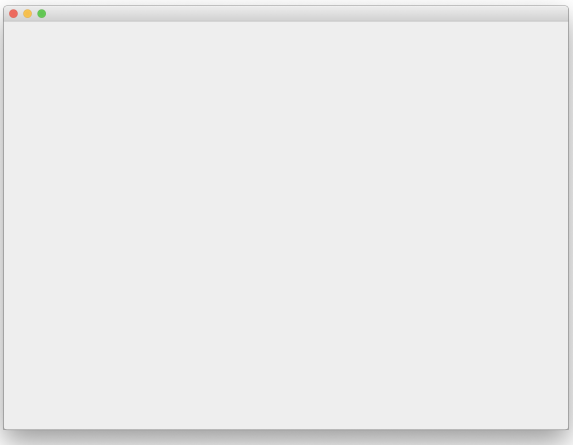
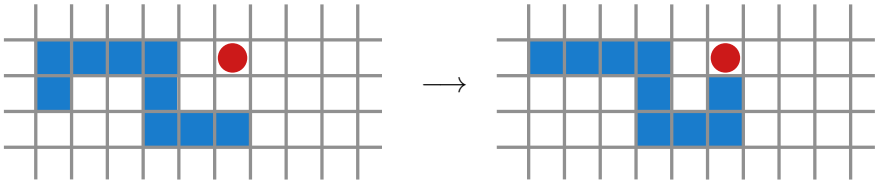


Abbildung 2: Leeres `AudGameWindow`. Das Fenster hat bei anderen Betriebssystemen evtl. eine andere Hintergrundfarbe.



Legen Sie in `SnakeGame` eine Konstante `SQUARE_SIZE` an, die die Kantenlänge eines Gitter-Quadrates in Pixeln speichern soll. Alle anderen Klassen des Projektes sollen auch ohne eine Instanz von `SnakeGame` auf diese Konstante zugreifen können. Weisen Sie ihr den Wert 16 zu!

g) Berechnen Sie im Konstruktor die Breite und Höhe des Spielfeldes in *Gitter-Koordinaten* (also wie viele Quadrate das Spielfeld breit und hoch ist) und speichern Sie diese Werte in den Attributen `width` und `height`.

3. **Point**

   Bevor wir mit der Implementierung der eigentlichen Spiel-Klassen beginnen, benötigen wir noch eine Hilfsklasse `Point`, die eine Position (x, y) auf dem Spielfeld speichert. Auf die beiden ganzzahligen Koordinaten soll von außen **nur** mit Hilfe von öffentlichen *getter*-Methoden zugegriffen werden können. Stellen Sie außerdem einen Konstruktor zur Verfügung, dem Anfangswerte für `x` und `y` übergeben werden können.

   Wichtig: Die Werte von x und y sind **keine** *Pixel-Koordinaten*, sondern *Gitter-Koordinaten*, sie beziehen sich also auf die Quadrate des Spielgitters! Pixel-Koordinaten werden ausschließlich dann verwendet, wenn es um das *Zeichnen* der Spiel-Oberfläche geht.

4. **Snake**

   Die Klasse `Snake` ist für die Darstellung der Schlange verantwortlich:

   a) Die Position der Schlange auf dem Spielfeld wird in einem Attribut `points` als Array von `Point`-Objekten gespeichert, jedes von der Schlange belegte Gitter-Quadrat entspricht also einem Eintrag in diesem Feld.

   b) Das Array enthält immer *mindestens* einen Punkt. Falls die Schlange gerade wächst, können Einträge am Ende leer sein (d. h. die *leere Referenz* `null` als Wert haben).

   c) Das Attribut `points` muss natürlich initialisiert werden. Erstellen Sie dafür einen Konstruk-

tor, dem drei `int`-Parameter übergeben werden:

i. `length` ist die Länge der Schlange und damit auch die Länge des Arrays `points`. Falls keine positive Zahl als Länge übergeben wurde, soll eine `IllegalArgumentException` geworfen werden.

ii. `x` und `y` sind die Gitter-Koordinaten des Anfangspunktes der Schlange. Legen Sie ein einzelnes entsprechendes `Point`-Objekt an und speichern Sie es am Beginn von `points`. Die weiteren Einträge bleiben für den Moment leer (`null`).

d) Legen Sie einen zweiten überladenen Konstruktor an, dem nur die Parameter `x` und `y` übergeben werden. Rufen Sie den ersten Konstruktor auf und übergeben Sie die Standardlänge 5.

e) Bevor wir uns um Wachstum und Bewegung kümmern, wollen wir die Schlange zuerst erst einmal statisch zeichnen:

i. Legen Sie dafür ein weiteres Attribut `color` vom Typ `java.awt.Color` an und initialisieren Sie es mit einer Farbe Ihrer Wahl. Diese Farbe soll zum Zeichnen der Schlange verwendet werden.

ii. Implementieren Sie dann eine Methode `void paint(AudGraphics g)`. Setzen Sie dort die Zeichenfarbe und zeichnen Sie für jeden Punkt aus `points` ein entsprechendes farbiges Quadrat.

> **Hinweis:** Verwenden Sie dabei die in `SnakeGame` definierte Kantenlänge der Gitter-Quadrate. Mit dieser Konstante können Sie auch die Gitter- in Pixel-Koordinaten umrechnen!

iii. Achten Sie beim Zeichnen darauf, dass Sie Einträge mit dem Wert `null` überspringen.

f) Legen Sie in `SnakeGame` nun ein Attribut für ein `Snake`-Objekt an. Ergänzen Sie den `SnakeGame`-Konstruktor so, dass dort eine neue Schlange in der Mitte des Spiel-Fensters erstellt wird.

g) Nun müssen Sie die Schlange nur in der Methode `paintGame()` zeichnen (⇝ `paintGame()`).

Ihr Programm sollte nun in etwa wie in Abbildung 3 aussehen:

> **Hinweis:** Sie wundern sich, warum nur ein Quadrat der Schlange gezeichnet wird, obwohl Sie wahrscheinlich ein `Snake`-Objekt mit Länge > 1 erzeugt haben? Keine Sorge, Sie haben nichts falsch gemacht! Die restlichen Schlangenelemente werden gezeichnet, sobald wir die Schlange in Bewegung setzen!

5. **Bewegung**

Nun zur vorhin erwähnten Bewegung:

a) Zunächst benötigen wir eine Möglichkeit, die Bewegungsrichtung der Schlange zu speichern. Legen Sie dafür in `Snake` eine öffentlich sichtbare *Enumeration* (⇝ siehe Vorlesungskapitel 5) namens `Direction` an, die als Aufzählung die Bewegungsrichtungen `RIGHT`, `DOWN`, `LEFT` und `UP` beinhaltet.

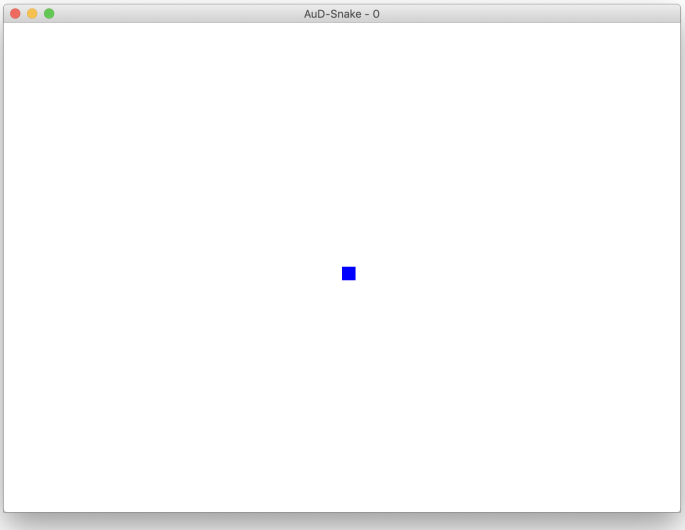


Abbildung 3: Spielfenster nach Erzeugen der Schlange

Hinweis: Im Gegensatz zur ebenfalls möglichen Variante, bei der Sie die einzelnen Bewegungsrichtungen als statische Konstanten anlegen würden, müssen Sie die Objekte innerhalb der *Enumeration* nicht explizit initialisieren. Java weist den Elementen im Hintergrund automatisch bei 0 beginnende, aufsteigende Werte zu.
Außerdem müssen Sie hier nicht immer überprüfen, ob ein `int`-Wert auch wirklich eine gültige *Direction* ist!

b) Die beim nächsten Update verwendete Richtung soll in der Schlange im Attribut `nextDirection` gespeichert werden. Initialisieren Sie es mit einem geeigneten Wert und stellen Sie außerdem *getter*- und *setter*-Methoden für das Attribut zur Verfügung.

Hinweis: Zwei Methoden von `Enumeration` könnten Ihnen im weiteren Verlauf der Aufgabe weiterhelfen:

- Sie bekommen den `int`-Wert, der einem bestimmten *Enum*-Objekt zu Grunde liegt, mittels `ordinal()`, also z. B. `LEFT.ordinal()`.
- Außerdem können Sie ein `int[]`-Array, das alle Elemente einer *Enum* beinhaltet, mittels `values()` bekommen, also z. B. `Direction.values()`.

c) Unsere Schlange wird sich nicht tatsächlich *kontinuierlich* fortbewegen, sondern wir werden ihre Position mehrmals pro Sekunde aktualisieren. Legen Sie dafür eine neue Methode `void step()` an, die dafür zuständig sein soll, einen solchen Schritt auszuführen:

i. Die Schlange bewegt sich, indem sie am Kopf um ein Feld wächst und am Ende das letzte Feld wegfällt. Demnach muss in `points[0]` (den Kopf) ein neuer Wert geschrieben werden, während die restlichen Punkte eine Position nach hinten rutschen (der bisher letzte Punkt in `points` wird nach dieser Operation nicht mehr von der Schlange besetzt und fällt weg).

ii. Informieren Sie sich zunächst über die Verwendung der Methode `System.arraycopy()` und implementieren Sie dann das Verschieben der hinteren Punkte des Arrays.

iii. Bestimmen Sie anschließend in Abhängigkeit von der Position des alten, ersten Punktes

und der Richtung `nextDirection` die neue Position des Kopfes und schreiben Sie sie an die erste Position des Arrays.

d) Im Hauptprogramm, der Klasse `SnakeGame`, muss `step()` jetzt natürlich noch aufgerufen werden. Die Häufigkeit soll über eine Konstante `STEP_TIME` eingestellt werden können, die die Zeit zwischen zwei Schritten in Millisekunden angibt. Ein guter Anfangswert könnten beispielsweise 100 ms (also 10 Updates pro Sekunde) sein.

e) Die eigentlichen Veränderungen können nun in der Methode `updateGame(long)` der Klasse `SnakeGame` durchgeführt werden, die von der bereitgestellten Oberklasse automatisch aufgerufen wird — allerdings in unregelmäßigen Abständen. Es könnte also sein, dass seit dem letzten Aufruf der Methode *kein* Schritt, *ein* Schritt oder (bei besonders kurzen `STEP_TIME`s oder besonders hoher Systemlast) sogar *mehrere* Schritte ausgeführt werden müssen.

f) Um herauszufinden, wie viele Schritte fällig sind, speichern Sie den (geplanten) Zeitpunkt (auch *Timestamp* genannt) des letzten Updates in einem `long`-Attribut `lastSnakeUpdate`. Im Konstruktor der Klasse können Sie die Methode `System.currentTimeMillis()` (die die aktuelle Systemzeit in Millisekunden liefert) verwenden, um dafür einen Anfangswert zu setzen.

g) Der Methode `updateGame(long)` wird als Parameter ebenfalls ein solcher Timestamp `time` übergeben, sodass Sie aus `time` und `lastSnakeUpdate` die verstrichene Zeit seit dem letzten Update berechnen können. Verwenden Sie `STEP_TIME`, um herauszufinden, wie oft die Schlange einen Schritt machen muss (⇝ Tipp: Schleife!). Rufen Sie entsprechend die `step`-Methode auf und erhöhen Sie nach jedem Update `lastSnakeUpdate` um `STEP_TIME`.

Hinweis: Wenn Sie das Programm jetzt starten, sollte sich die Schlange je nach gewählter Bewegungsrichtung – wie in Abbildung 4 – bewegen!

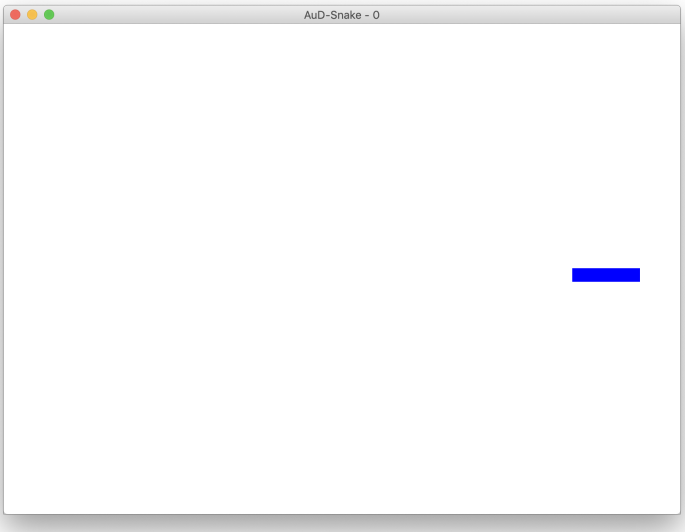


Abbildung 4: Spielfenster mit Schlangenbewegung

6. **Steuerung**

Damit der Spieler die Bewegung der Schlange beeinflussen kann, soll das Programm jetzt noch auf das Drücken der Pfeiltasten auf Ihrer Tastatur reagieren.

a) Hierfür ist die Methode `handleInput(int)` gedacht: Sie bekommt als Parameter den Tasten-Code einer gedrückten Taste übergeben.

b) Die verschiedenen Tasten-Codes sind als öffentliche, statische Integer-Konstanten in der Klasse `java.awt.event.KeyEvent` definiert. Leider können wir diese aufgrund von Beschrän-kungen im StudOn nicht benutzen. Wir bieten Ihnen in `AuDGameWindow` eine Alternative, die sie mit `KeyEvent.VK_RIGHT`, `KeyEvent.VK_DOWN`, `KeyEvent.VK_LEFT` und `KeyEvent.VK_UP`, nutzen können, um die gewünschte nächste Richtung der Schlange zu setzen. Verwenden Sie unbedingt die in `Direction` definierten Richtungen!

c) Um zu vermeiden, dass die Richtung der Schlange durch Tastendruck direkt umgekehrt wird (z. B. bei einem Wechsel von `RIGHT` zu `LEFT`, was zu einer sofortigen Kollision mit sich selbst führen würde), können Sie in der Klasse `Snake` in einem weiteren Attribut `lastDirection` die im letzten Aufruf von `step()` eingeschlagene Richtung speichern. Falls nun `setNextDirection` eine falsche Richtung übergeben wird, können Sie dort die Anweisung einfach ignorieren.

Testen Sie das Ergebnis!

7. **GameItem**

Für die übrigen, statischen Spiel-Objekte lohnt es sich, eine gemeinsame, abstrakte Basisklasse `GameItem` anzulegen.

a) Im Konstruktor sollen der Klasse die Koordinaten `int x` und `int y` übergeben werden, die als `Point`-Objekt in einem Attribut `position` gespeichert werden. Von außen soll der Zugriff auf dieses Attribut mittels einer *getter*-Methode ermöglicht werden.

b) Alle von `GameItem` abgeleiteten Klassen müssen eine Methode `paint`-Methode zur Verfügung stellen, der wie schon in `Snake` ein `Graphics`-Objekt übergeben wird. Legen Sie eine entsprechende abstrakte Methode an!

8. **Brick**

Die erste von `GameItem` abgeleitete Klasse soll `Brick` sein. Mit Objekten dieser Klasse soll eine Mauer definiert werden, um zu verhindern, dass die Schlange das Spielfeld verlassen kann.

a) Ein `Brick` soll immer genau ein Gitter-Quadrat ausfüllen und entsprechend als farbiges Quadrat gezeichnet werden.

b) `Brick` soll die gleichen Konstruktor-Parameter unterstützen wie `GameItem`.

Hinweis: Denken Sie daran, dass Konstruktoren nicht vererbt werden und Sie auch den Konstruktor der Eltern-Klasse manuell aufrufen müssen!

c) Legen Sie nun in `SnakeGame` in Abhängigkeit von der Spielfeldgröße (siehe oben) eine durchgehende Mauer aus `Brick`-Objekten an den *Rändern* des Spielfeldes an. Speichern Sie sie als eindimensionales Array in einem Attribut `wall` und sorgen Sie dafür, dass die Mauer in der `paintGame`-Methode gezeichnet wird!

d) Ihr Programm sollte nun wie in Abbildung 5 aussehen.

9. **Kollisionen**

Wenn Sie nun spielen wollen, werden Sie feststellen, dass Sie sich einfach durch die Wand bewegen können. Um dies zukünftig zu verhindern, müssen wir eine Kollisionsbehandlung für Objekte erstellen.

a) Ergänzen Sie dafür die Klasse `Snake` um zwei Methoden `boolean collidesWith(GameItem item)` und `boolean collidesWith(int x, int y)`. Falls irgendein Teil der Schlange die Koordinaten (x, y) bzw. dieselben Koordinaten wie das übergebene `GameItem` besetzt, gibt

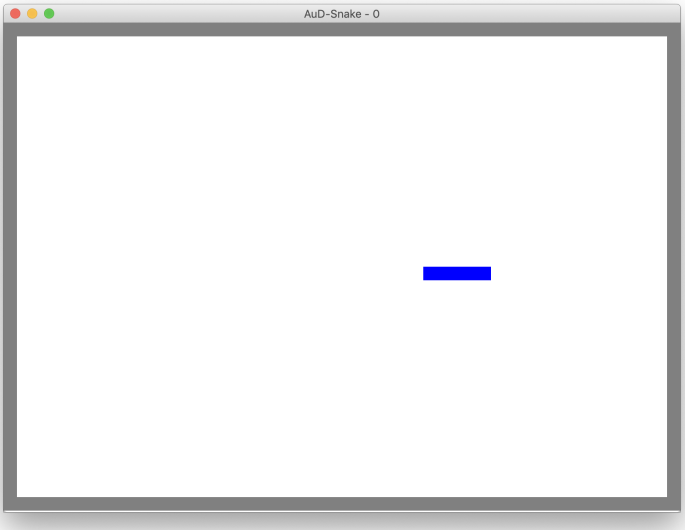


Abbildung 5: Spielfeld mit Mauerumrandung

es einen Zusammenstoß und die Methoden sollen `true` zurückgeben, andernfalls `false`.

Hinweis: Die Methode `collidesWith` wurde *überladen*, Sie müssen die Kollisionsbehandlung also nur in einer der beiden Methoden implementieren und diese aus der anderen heraus aufrufen. Überlegen Sie sich, welche Variante am sinnvollsten ist, also welche Methode die Kollisionsbehandlung implementieren und welche Methode dann die andere aufrufen soll!

b) Erstellen Sie in `SnakeGame` eine Hilfsmethode `checkCollisions`, die Sie nach jedem Schritt aus `updateGame` aufrufen.

c) Überprüfen Sie für jeden `Brick` in der Mauer, ob es einen Zusammenstoß mit der Schlange gibt. Falls ja, rufen Sie die (geerbte) Methode `stop()` auf und zeigen *anschließend* eine Dialog-Box an, in der Sie dem Spieler mitteilen, dass er verloren hat.

Verwenden Sie dazu die Methode `showDialog(String)`, die Sie von `AudGameWindow` geerbt haben. Die Nachricht soll wie in wie in Abbildung 6 angezeigt werden. Beenden Sie anschließend die Methode.

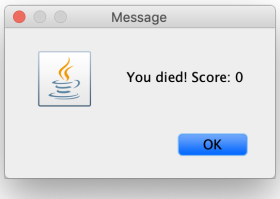


Abbildung 6: Dialog nach verlorenem Spiel

Testen Sie die Kollisionserkennung!

d) Außerdem darf die Schlange auch nicht mit sich selbst zusammenstoßen. Erstellen Sie dazu in der Klasse `Snake` eine Methode `boolean collidesWithSelf()`, die das überprüft und wieder entsprechend `true` oder `false` zurückgibt.

Ergänzen in `SnakeGame` die Methode `checkCollisions` um diese zusätzliche Kollisionserkennung.

10. **Apple**

Ein letztes Spielelement fehlt nun noch: Die Schlange muss zufällig auf dem Spielfeld platzierte Äpfel einsammeln. Wenn sie einen Apfel gefressen hat, wächst sie und bekommt dafür Punkte.

a) Legen Sie eine Klasse `Apple` an, die ebenfalls von `GameItem` erbt. Denken Sie daran, wie schon in `Brick` den Konstruktor der Elternklasse aufzurufen.

b) Den Apfel können Sie beispielsweise als roten ausgefüllten Kreis darstellen. Zeichnen Sie den Apfel in `paintGame`.

Hinweis: Beachten Sie, dass die `Graphics`-API nur Methoden zum Zeichnen von Ovalen besitzt!

c) Die Punkte für das Einsammeln eines Apfels sollen mit jedem Apfel um eins erhöht werden. Für den ersten Apfel gibt es einen Punkt, für den zweiten zwei, usw. Legen Sie dafür ein unveränderliches Attribut `VALUE` an, das über eine *getter*-Methode ausgelesen werden kann. Weisen Sie diesem Attribut im Konstruktor einen Wert zu, sodass automatisch das erste erzeugte `Apple`-Objekt den Wert 1 erhält, das zweite den Wert 2, etc.

d) Legen Sie in `SnakeGame` eine Hilfsmethode `createNewApple()` an, die an einer zufälligen Stelle innerhalb des Spielfeldes einen Apfel erstellt und das Objekt in einem Attribut `apple` speichert. Die Position des Apfels darf nicht schon beim Anlegen mit der Schlange kollidieren. Überprüfen Sie dies, **bevor** Sie das `Apple`-Objekt erzeugen! Erzeugen Sie gegebenenfalls so lange einen neuen Apfel, bis diese Bedingung erfüllt ist.

e) Rufen Sie im Konstruktor die neue Methode zur Apfelerzeugung auf.

11. **Schlangenwachstum und Punktestand**

Damit die Schlange wachsen kann, benötigen wir noch eine letzte Methode in `Snake`.

a) Speichern Sie die Anzahl an Quadraten, um die die Schlange pro Apfel wachsen soll, in einer Konstante `GROW_AMOUNT`.

b) Die Methode `grow` soll einen `int`-Parameter übergeben bekommen, der die Anzahl der Felder angibt, um die die Schlange wachsen soll. Ist der übergebene Wert keine positive Zahl oder zu klein, so soll eine `IllegalArgumentException` mit einer entsprechenden Meldung geworfen werden.

c) Legen Sie dementsprechend ein neues `Point`-Array mit der richtigen Länge an und kopieren Sie das alte Feld an den Anfang des neuen. Aktualisieren Sie anschließend das Attribut.

d) Erweitern Sie `checkCollisions` um die Überprüfung, ob die Schlange den Apfel erreicht hat. Falls ja, rufen Sie die `grow`-Methode der Schlange auf (z. B. mit dem Wert 5) und erzeugen einen neuen Apfel!

e) Jetzt müssen nur noch die Punkte gezählt werden. Addieren Sie jedes Mal, wenn die Schlange einen Apfel einsammelt, den in diesem Apfel gespeicherten `value` zum Gesamtpunktestand `score`.

f) Sorgen Sie mit Hilfe der bereits erwähnten Methode `setTitle` dafür, dass in der Titelleiste des Fensters immer der aktuelle Punktestand angezeigt wird – im Format

`AuD-Snake - Score: ⟨score⟩`

12. Ihr Programm sollte nun so aussehen:

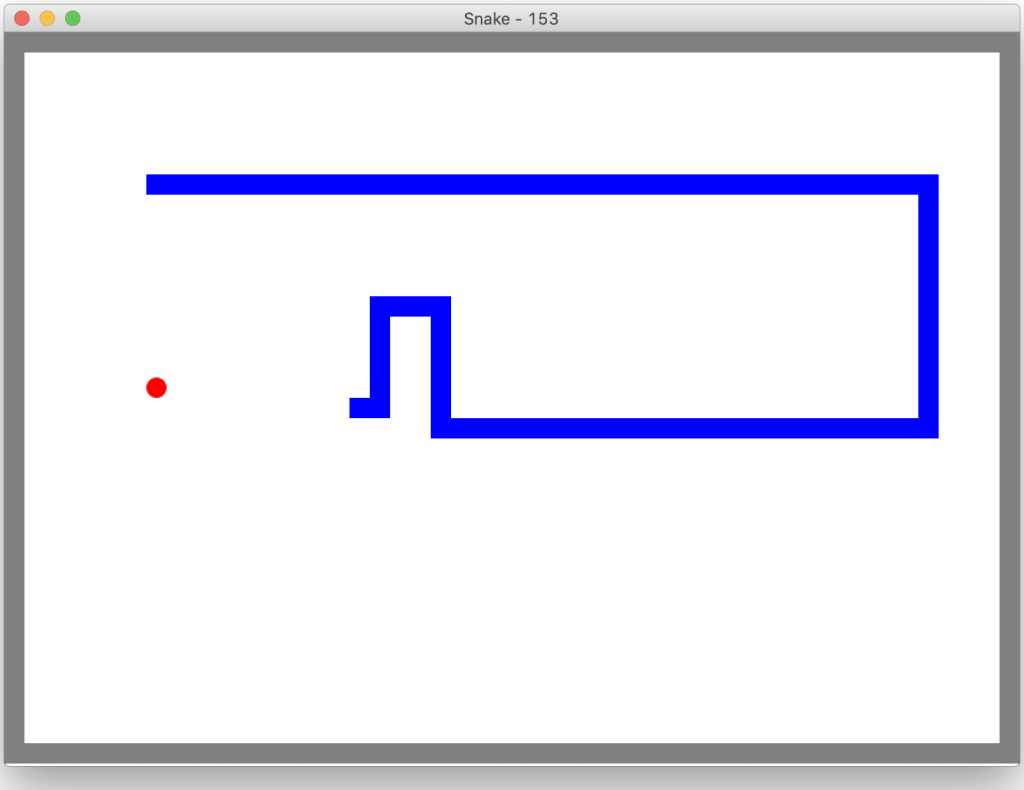


Abbildung 7: Fertiges Spiel mit wachsender Schlange und Äpfeln

13. Testen Sie das Spiel **ausführlich** und geben Sie anschließend die Dateien `Apple.java`, `Brick.java`, `GameItem.java`, `Point.java`, `SnakeGame.java` und `Snake.java` ab!

Achtung: Achten Sie darauf, dass Attributsnamen, Sichtbarkeiten und Schnittstellen **exakt** eingehalten wurden!

Sollte Ihr Programm nicht übersetz- bzw. ausführbar sein, wird die Lösung mit 0 Punkten bewertet. Stellen Sie also sicher, dass IntelliJ IDEA keine Fehler in Ihrem Programm anzeigt, Ihr Programm übersetz- und ausführbar ist sowie die in der Aufgabenstellung vorgegebenen Namen und Schnittstellen *exakt* eingehalten werden.